

Datapath

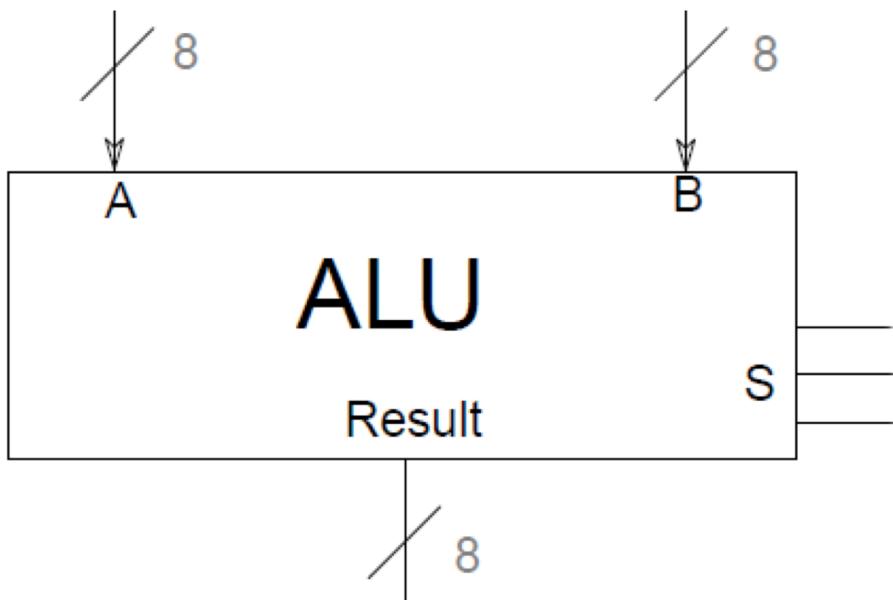
Arquitectura de Computadores – IIC2343

Datapath

El componente del procesador, o CPU, que realiza operaciones aritméticas

La ruta que conecta a los registros con la ALU, y por la cual viajan los datos

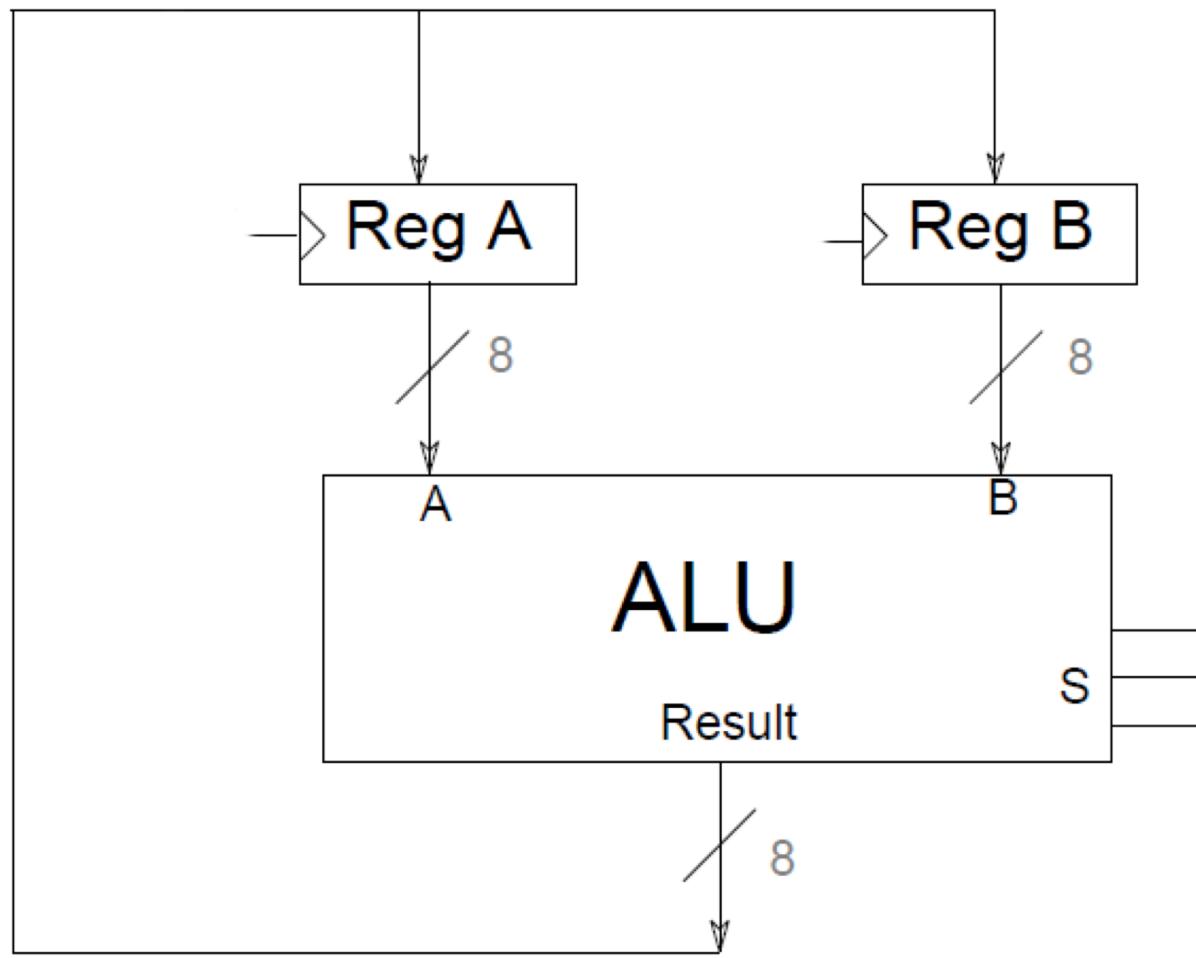
Dónde estamos



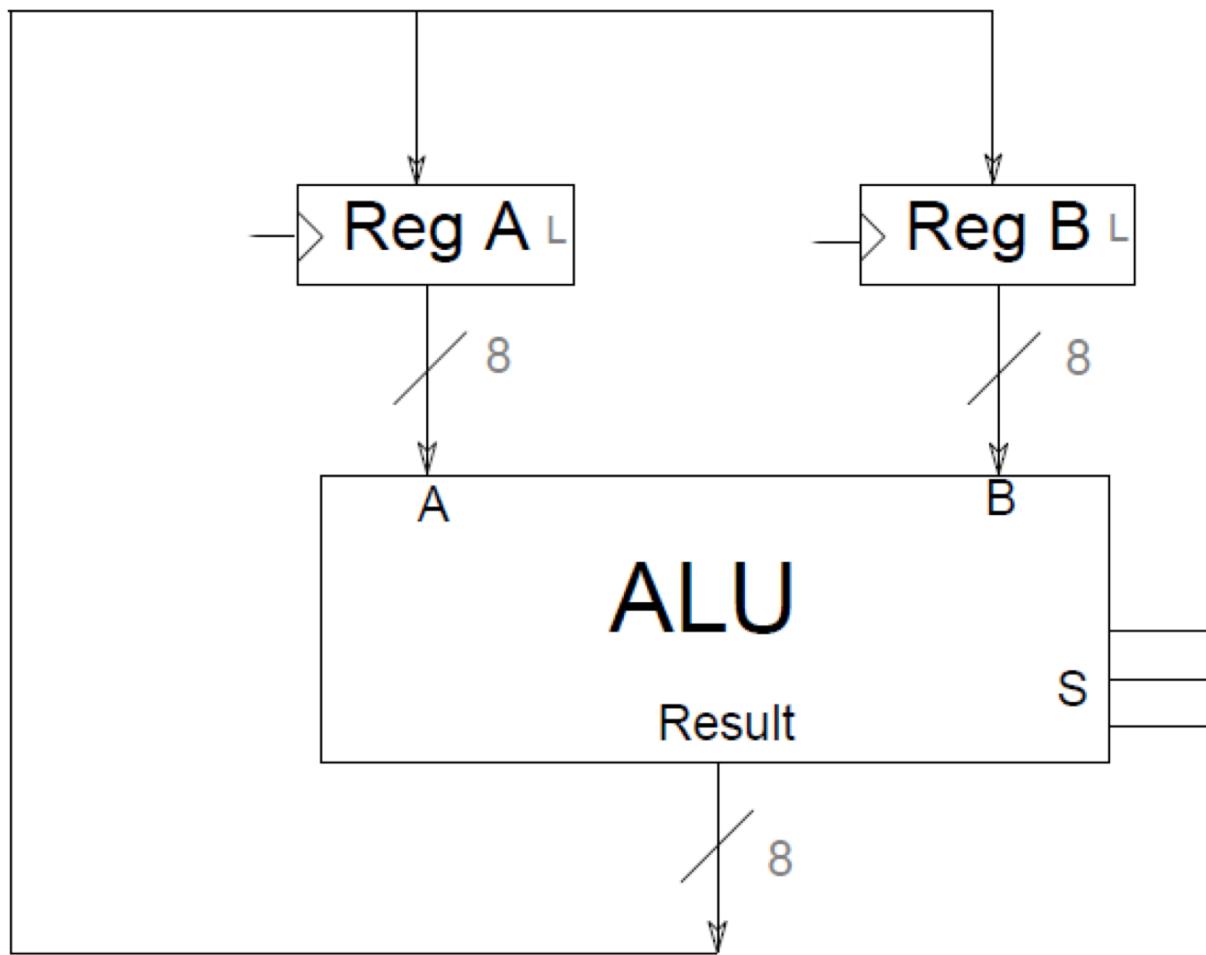
S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

Las entradas A y B provienen de **registros** de la CPU

La salida *Result* va a parar a esos mismos registros

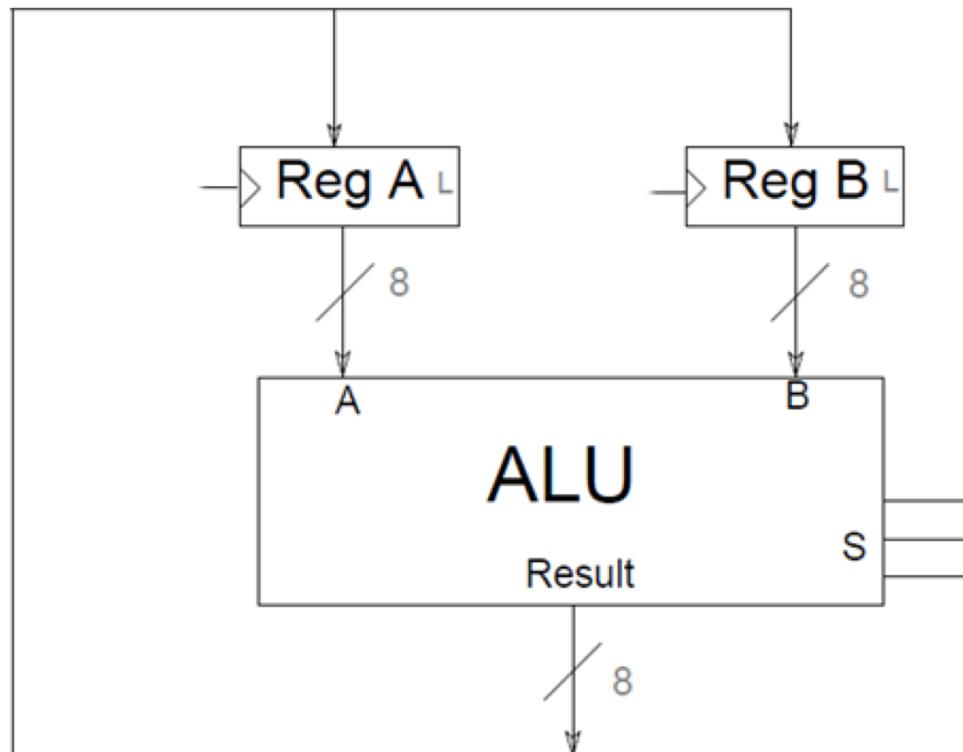


Agregamos las señales de control L_A y L_B para controlar la escritura —o actualización— de los valores— de los registros



Las diferentes combinaciones de valores de las cinco señales de control especifican qué acciones puede ejecutar este circuito:

- qué operación se ejecuta — S_0, S_1 y S_2
- a dónde va a parar el resultado — L_A y L_B



la	lb	s2	s1	s0	operación
1	0	0	0	0	A=A+B
0	1	0	0	0	B=A+B
1	0	0	0	1	A=A-B
0	1	0	0	1	B=A-B
1	0	0	1	0	A=A and B
0	1	0	1	0	B=A and B
1	0	0	1	1	A=A or B
0	1	0	1	1	B=A or B
1	0	1	0	0	A=notA
0	1	1	0	0	B=notA
1	0	1	0	1	A=A xor B
0	1	1	0	1	B=A xor B
1	0	1	1	0	A=shift left A
0	1	1	1	0	B=shift left A
1	0	1	1	1	A=shift right A
0	1	1	1	1	B=shift right A

P.ej., si a partir de los valores 0 y 1 almacenados en los registros A y B, respectivamente, ejecutamos las seis acciones que se muestran en la columna de la izquierda, los registros quedan con los valores que se muestran en las columnas de la derecha

la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A = A + B$	1	1
0	1	0	0	0	$B = A + B$	1	2
1	0	0	0	0	$A = A + B$	3	2
0	1	0	0	0	$B = A + B$	3	5
1	0	0	0	0	$A = A + B$	8	5
0	1	0	0	0	$B = A + B$	8	13

Cada combinación de valores de las señales de control es una **instrucción**

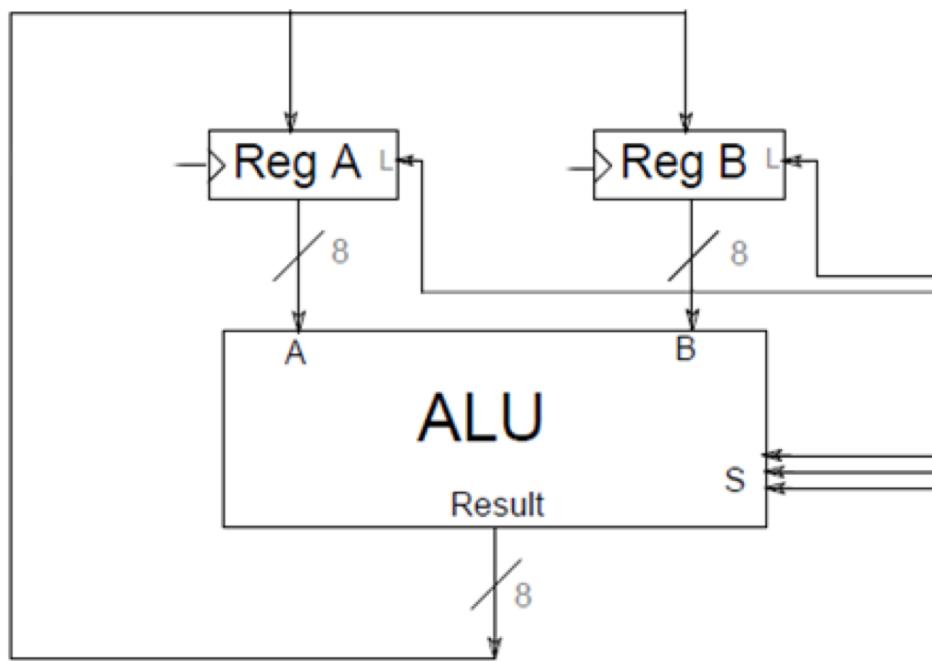
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A=A+B$	1	1
0	1	0	0	0	$B=A+B$	1	2
1	0	0	0	0	$A=A+B$	3	2
0	1	0	0	0	$B=A+B$	3	5
1	0	0	0	0	$A=A+B$	8	5
0	1	0	0	0	$B=A+B$	8	13

Cada combinación de valores de las señales de control es una **instrucción**

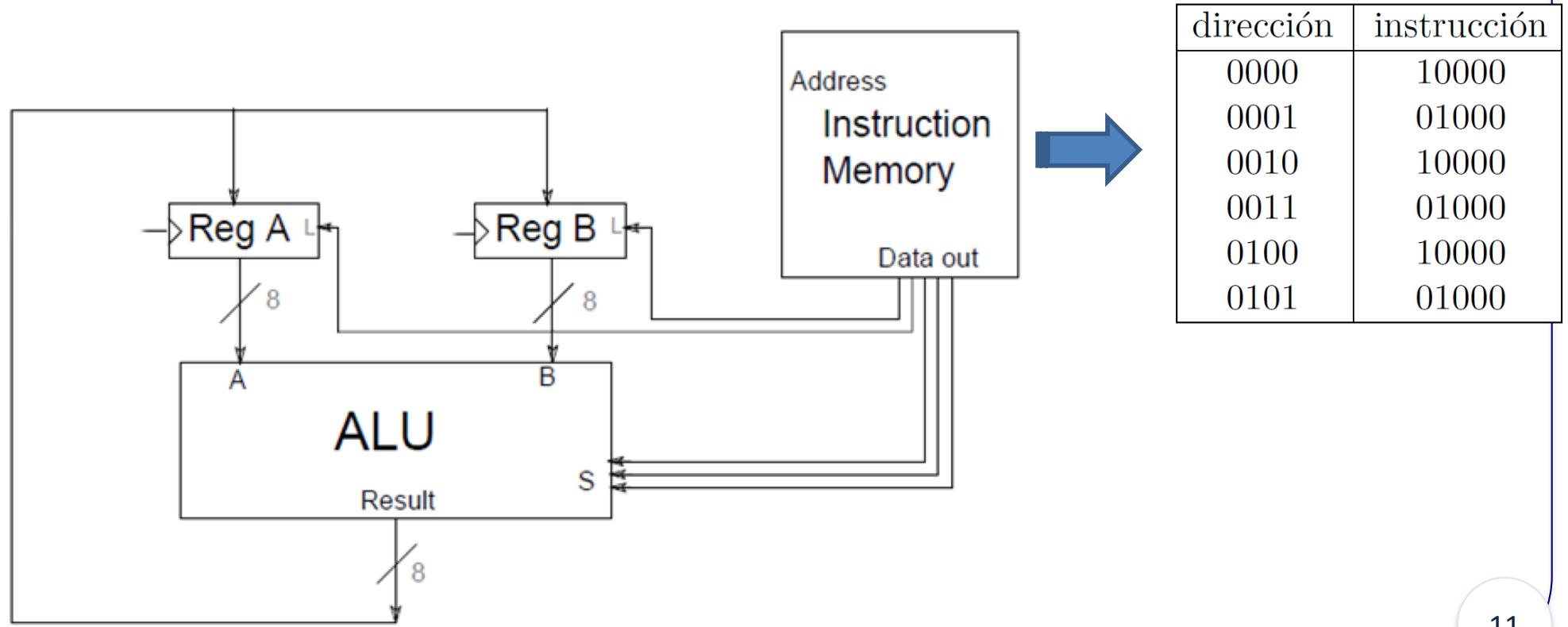
... y una secuencia de instrucciones es un **programa**

la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A = A + B$	1	1
0	1	0	0	0	$B = A + B$	1	2
1	0	0	0	0	$A = A + B$	3	2
0	1	0	0	0	$B = A + B$	3	5
1	0	0	0	0	$A = A + B$	8	5
0	1	0	0	0	$B = A + B$	8	13

Los computadores von Neumann se caracterizan porque el programa —la secuencia de instrucciones— está almacenado en el mismo computador ...

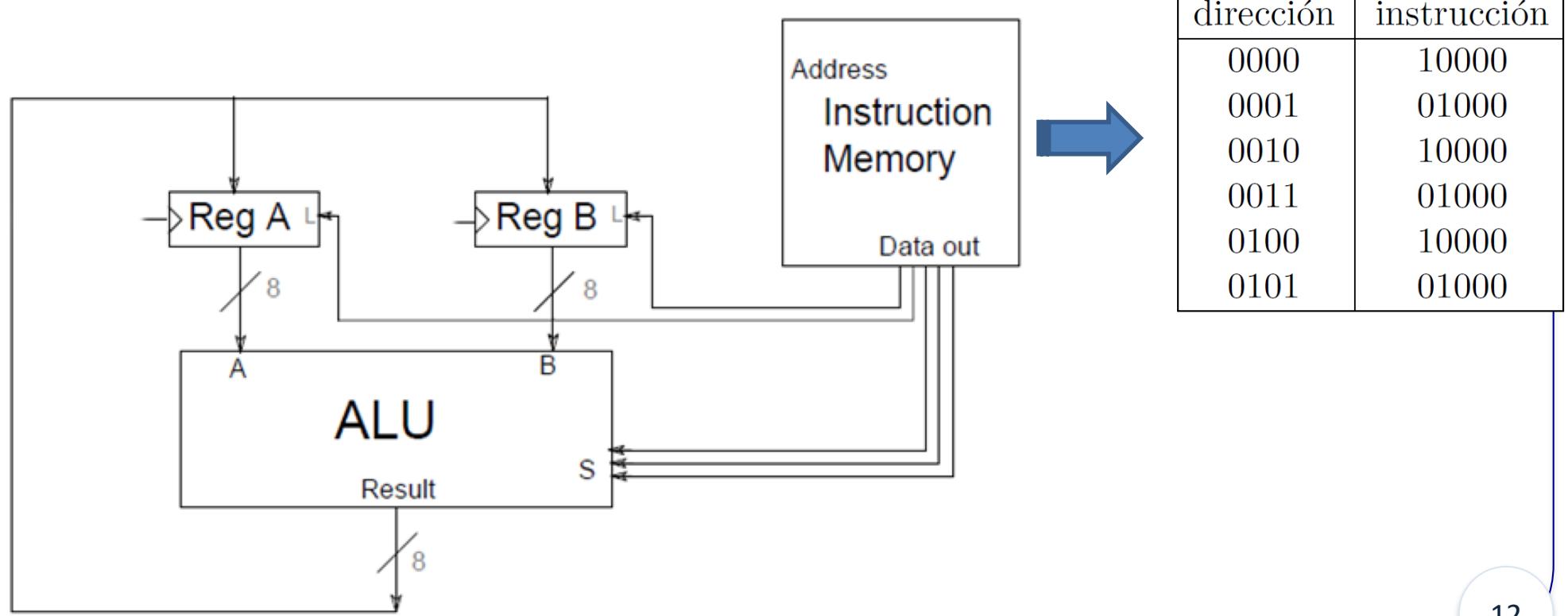


... en una **memoria**, p.ej., una memoria de tipo **ROM** (*read-only memory*)



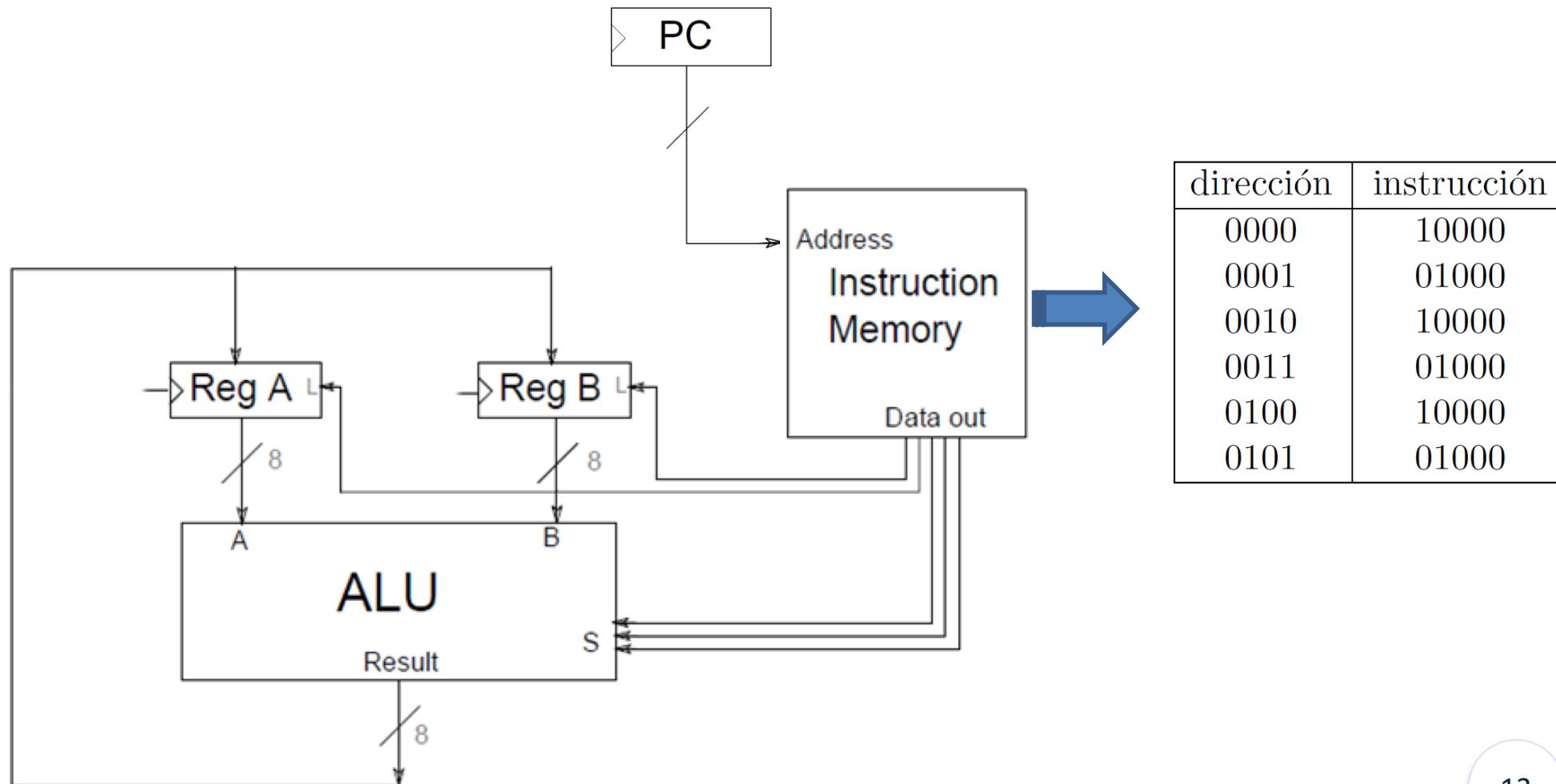
Necesitamos que la lectura/ejecución de las instrucciones sea secuencial

... es decir, que por *Data out* vayan saliendo las instrucciones en el mismo orden en que están escritas en el programa



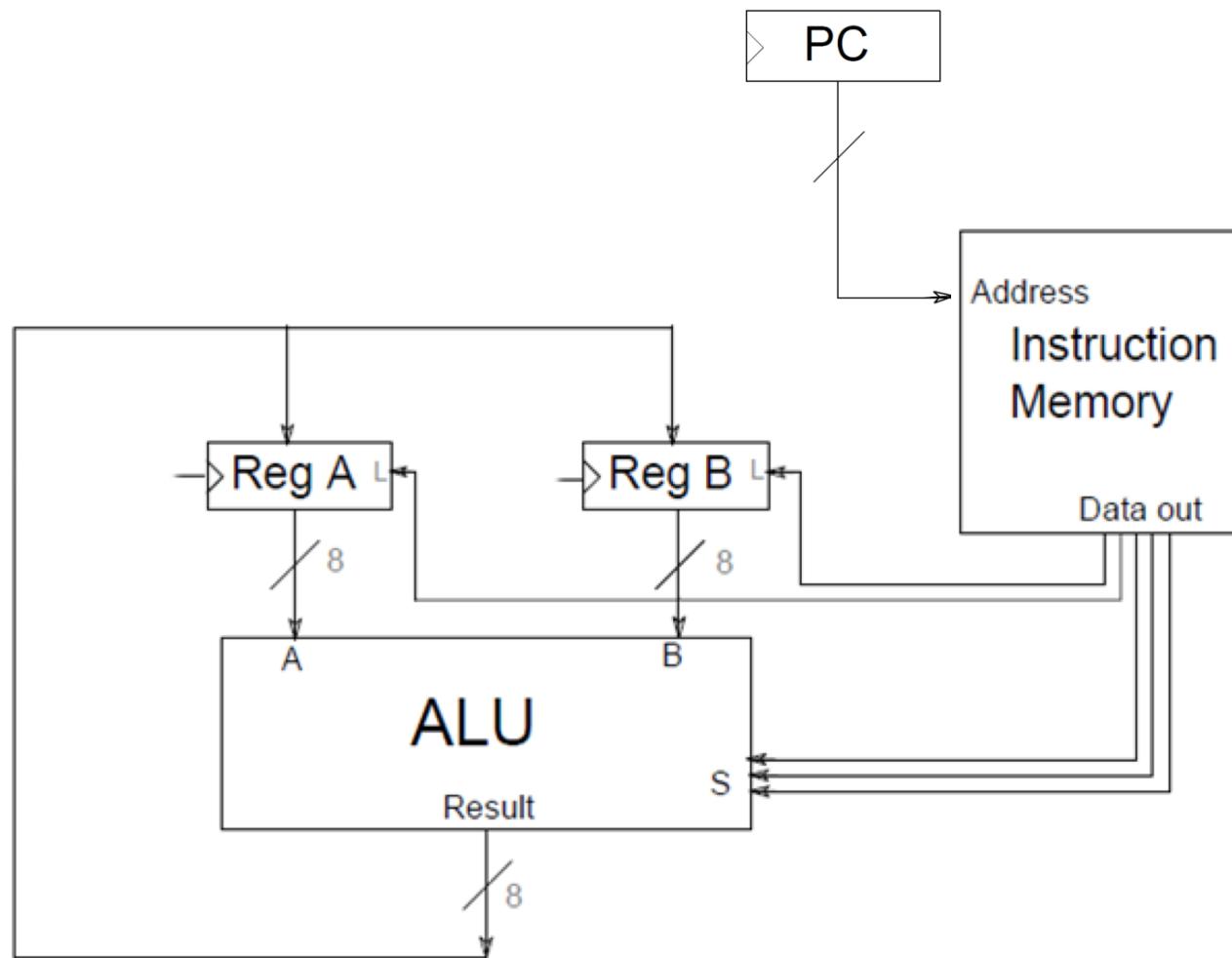
PC —o *program counter* (o *instruction pointer*)— es un registro que almacena una dirección de memoria,

... tal que al conectarse a la entrada *Address* de la memoria, la instrucción que está en esa dirección es seleccionada y puesta en *Data out*



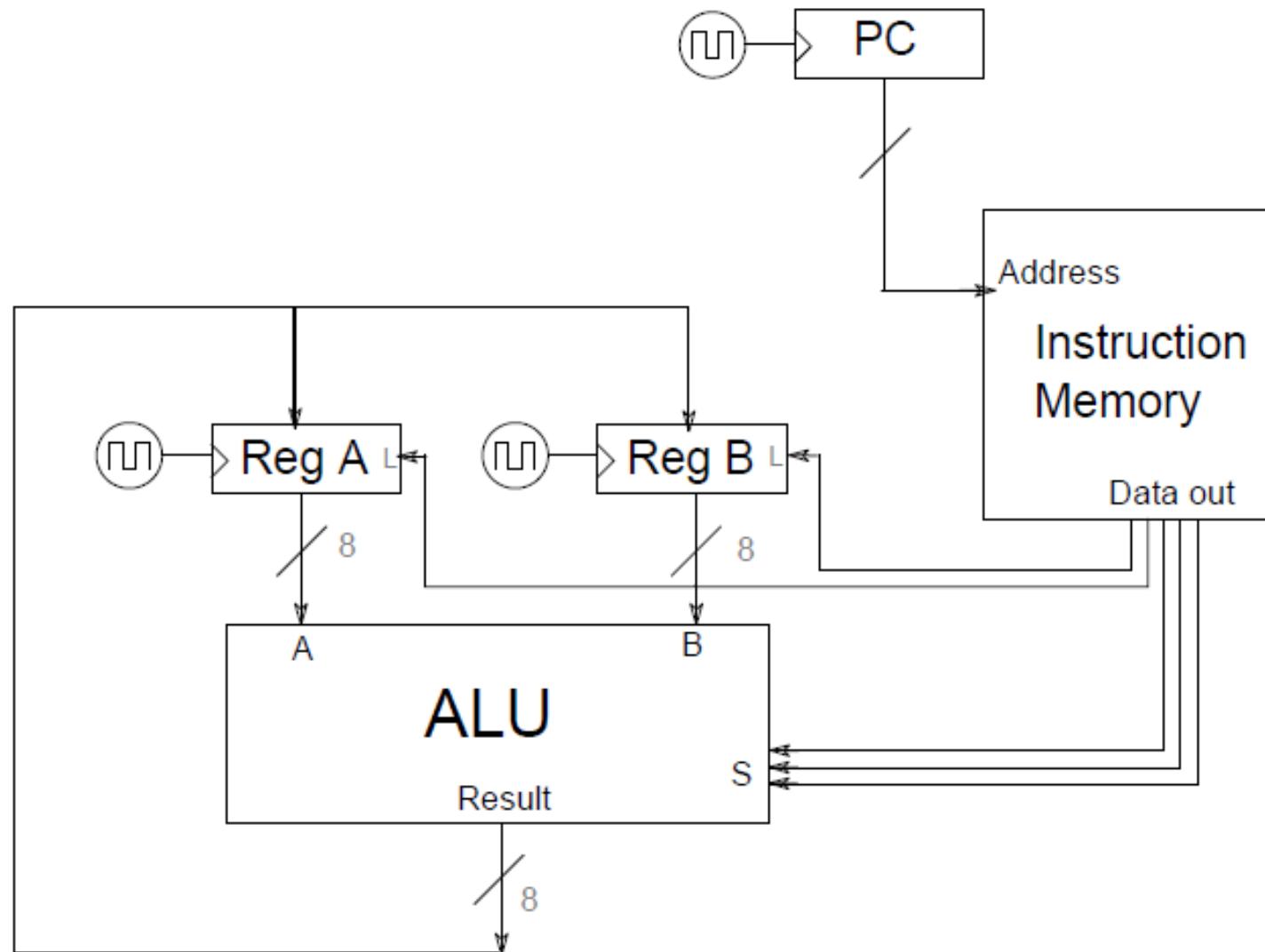
Finalmente, necesitamos que el registro *PC* vaya incrementando automáticamente su contenido,

... para que el programa se ejecute por completo sin más intervención nuestra que a la partida

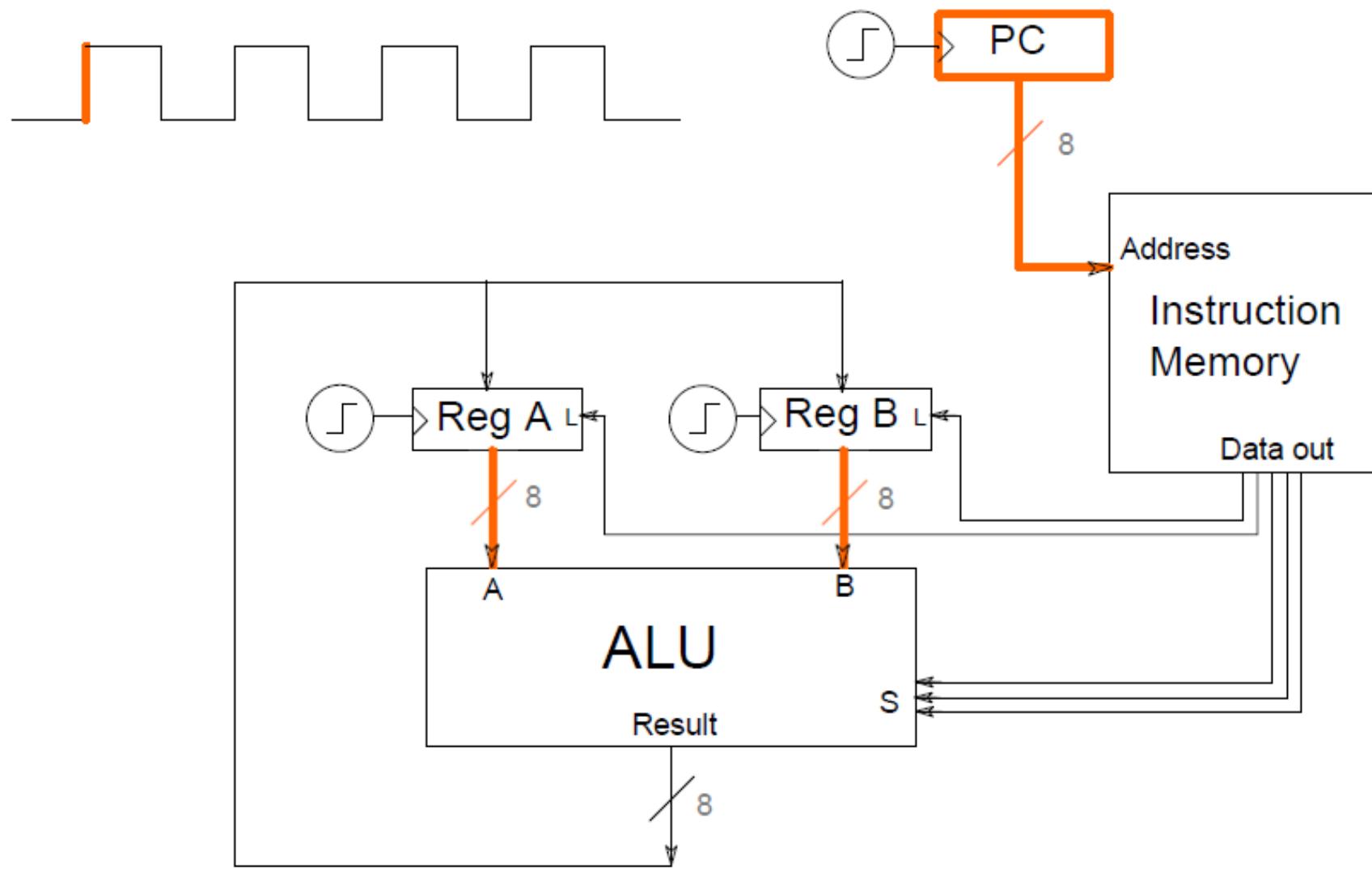


... y que todas las acciones individuales ocurran sincrónicamente:

Incluimos un *reloj* (*clock*)

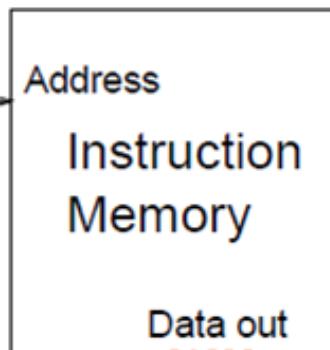


Reloj: Circuito que emite una serie de pulsos con un ancho preciso y un intervalo preciso entre pulsos consecutivos, y cuya frecuencia es controlada por un oscilador de cristal



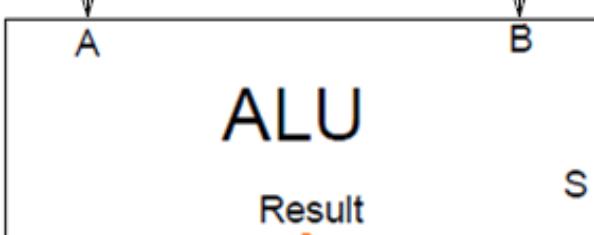
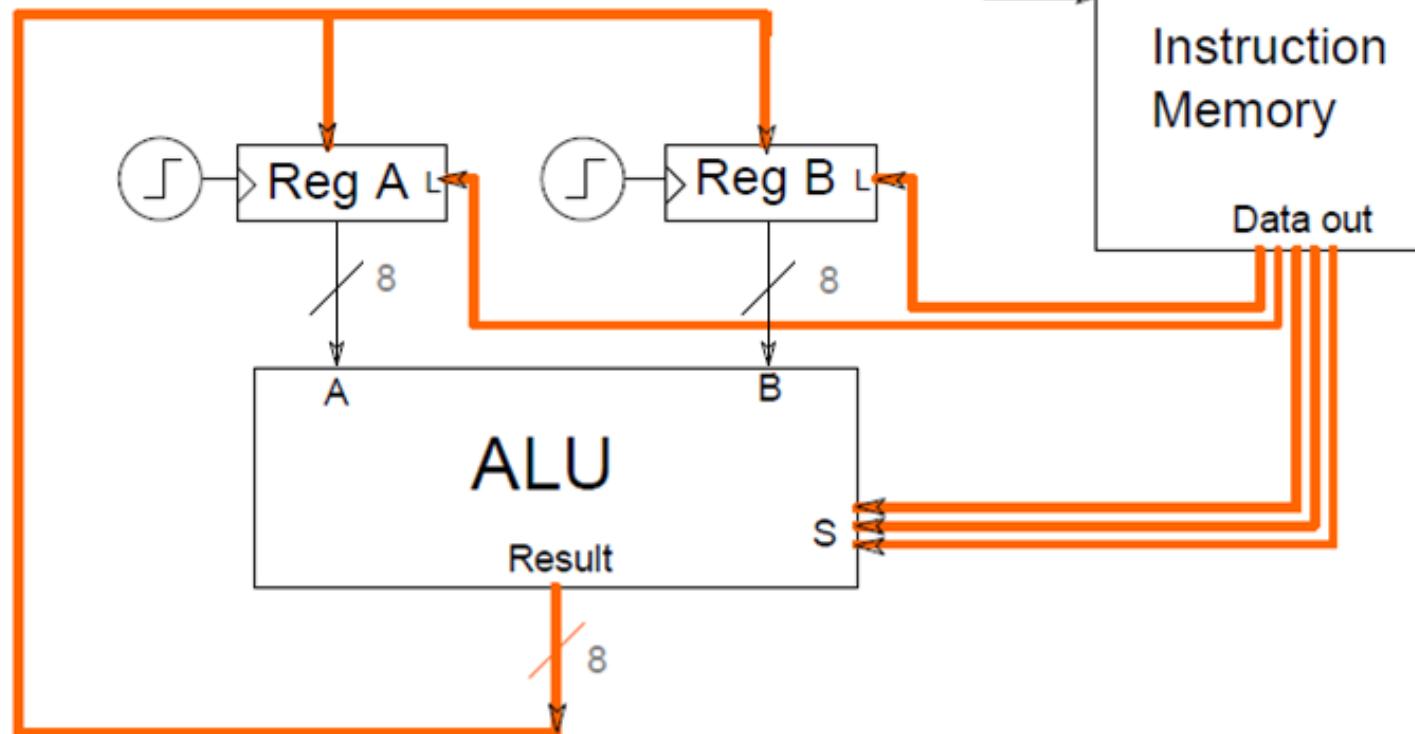


8



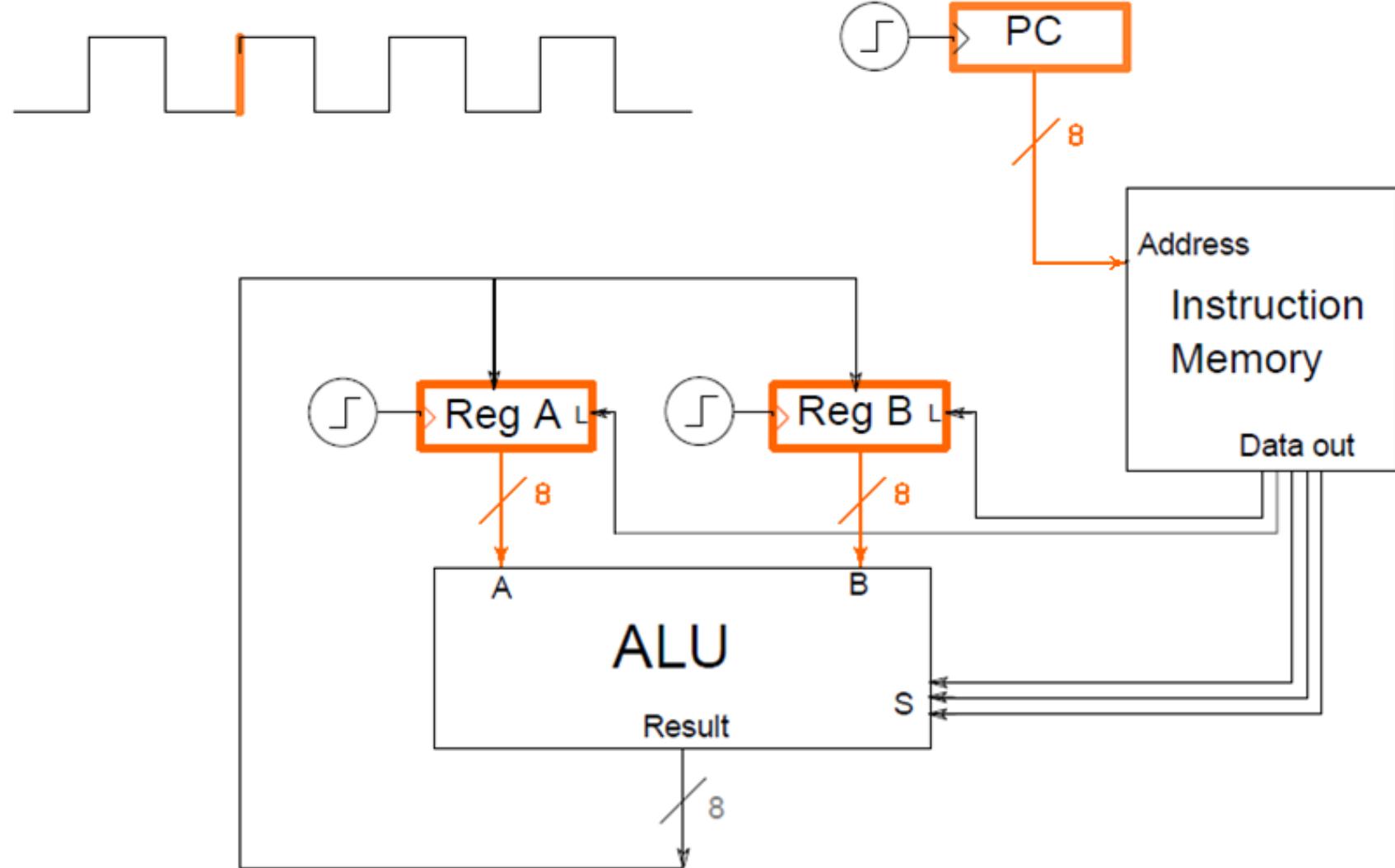
Address
Instruction
Memory

Data out



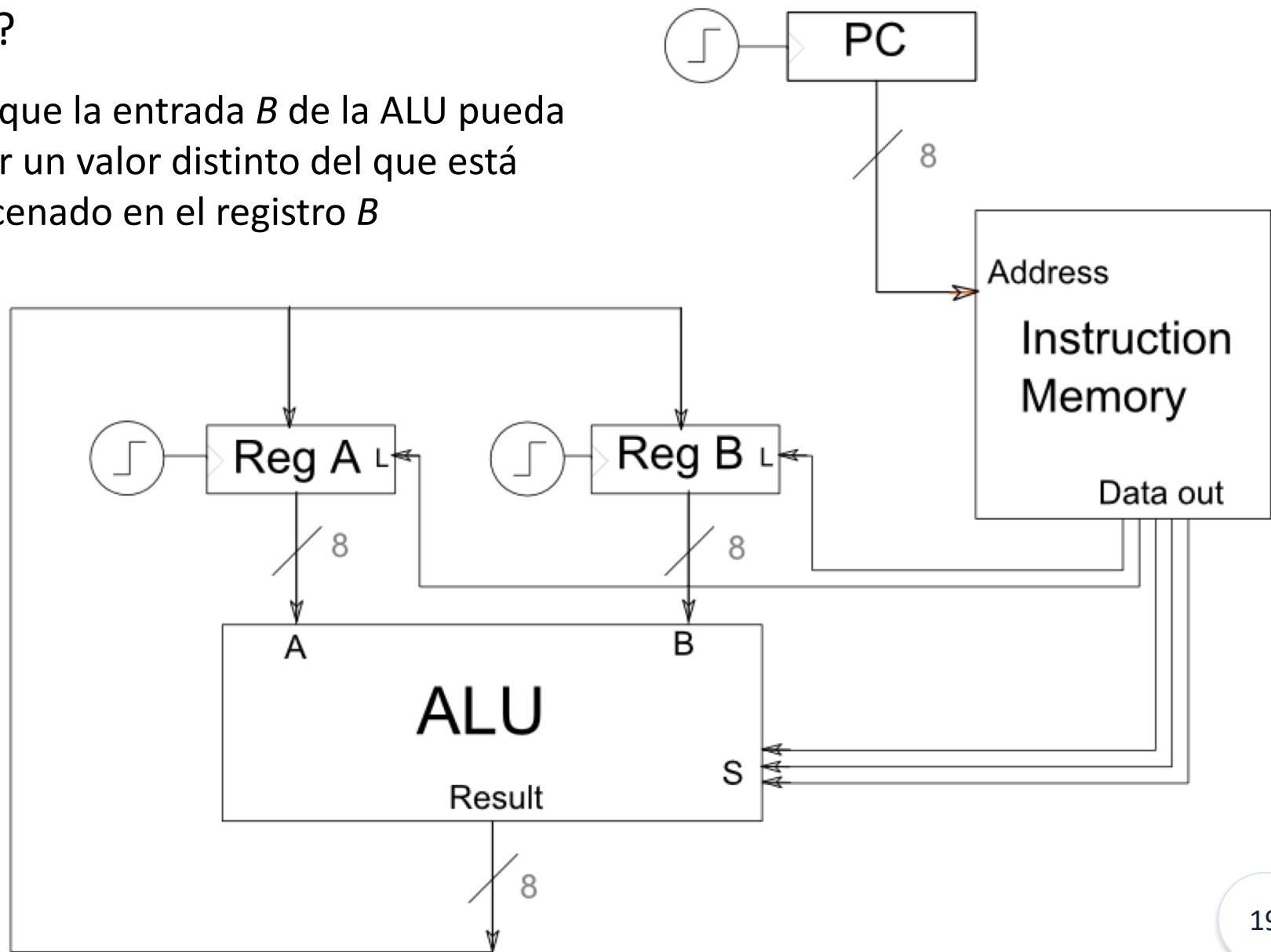
Result

8



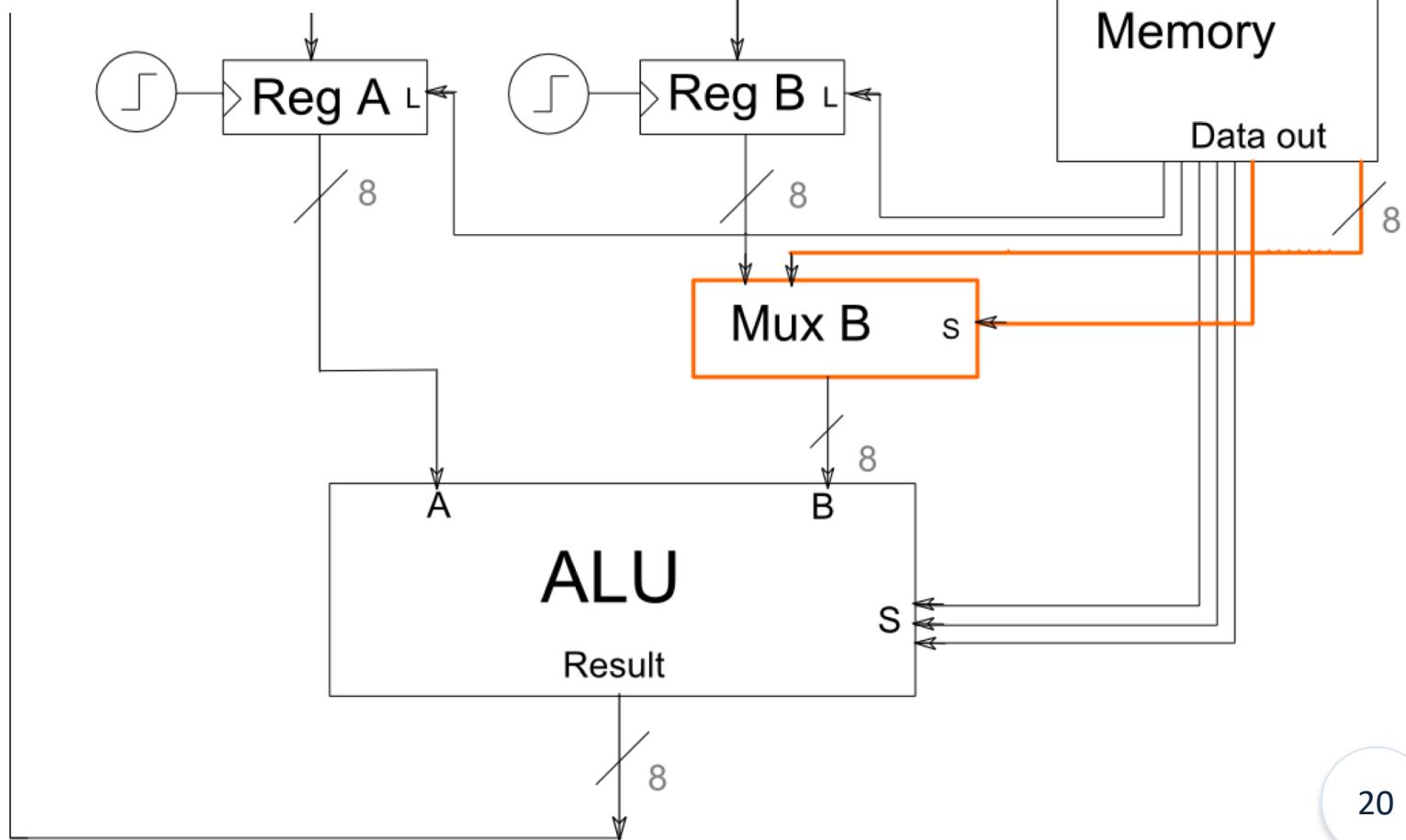
¿Cómo podemos hacer para independizarnos de los valores iniciales en los registros?

- p.ej., que la entrada *B* de la ALU pueda recibir un valor distinto del que está almacenado en el registro *B*



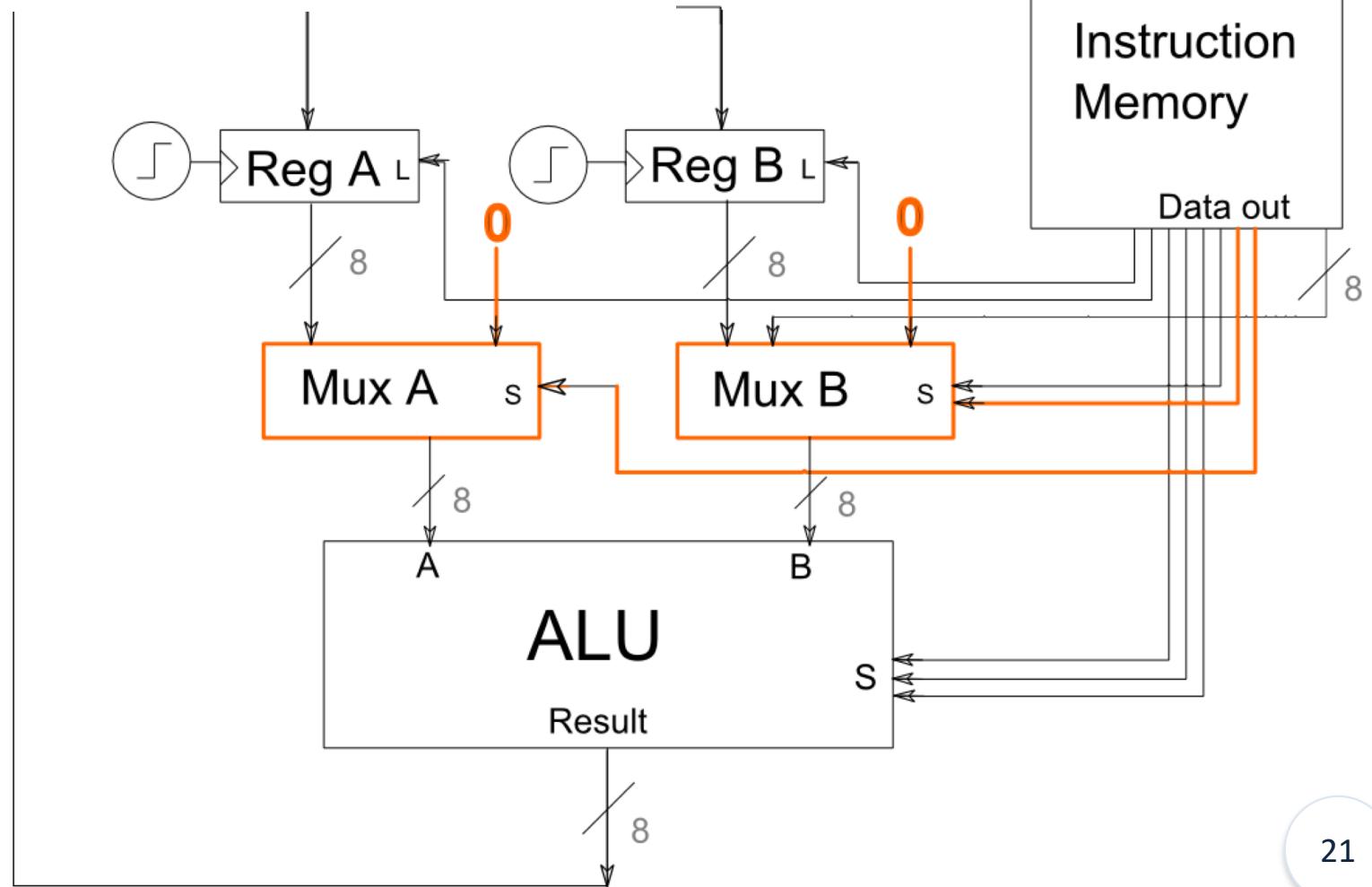
Agregamos a las instrucciones un campo de 8 bits para representar *literales*

... y usamos un multiplexor para decidir si el valor que va a la entrada *B* de la ALU es el contenido del registro *B* o el literal que viene en la instrucción



Extendemos el uso de los multiplexores para permitir poner el valor 0 (muy común) en las entradas A o B

Estos dos casos, y el de la diapositiva anterior, exigen señales de control adicionales



Hay 8 señales de control,
permitiendo $2^8 = 256$
instrucciones posibles

... pero solo tenemos 28
instrucciones

¿Cómo podemos ahorrar
espacio en la memoria
de instrucciones?

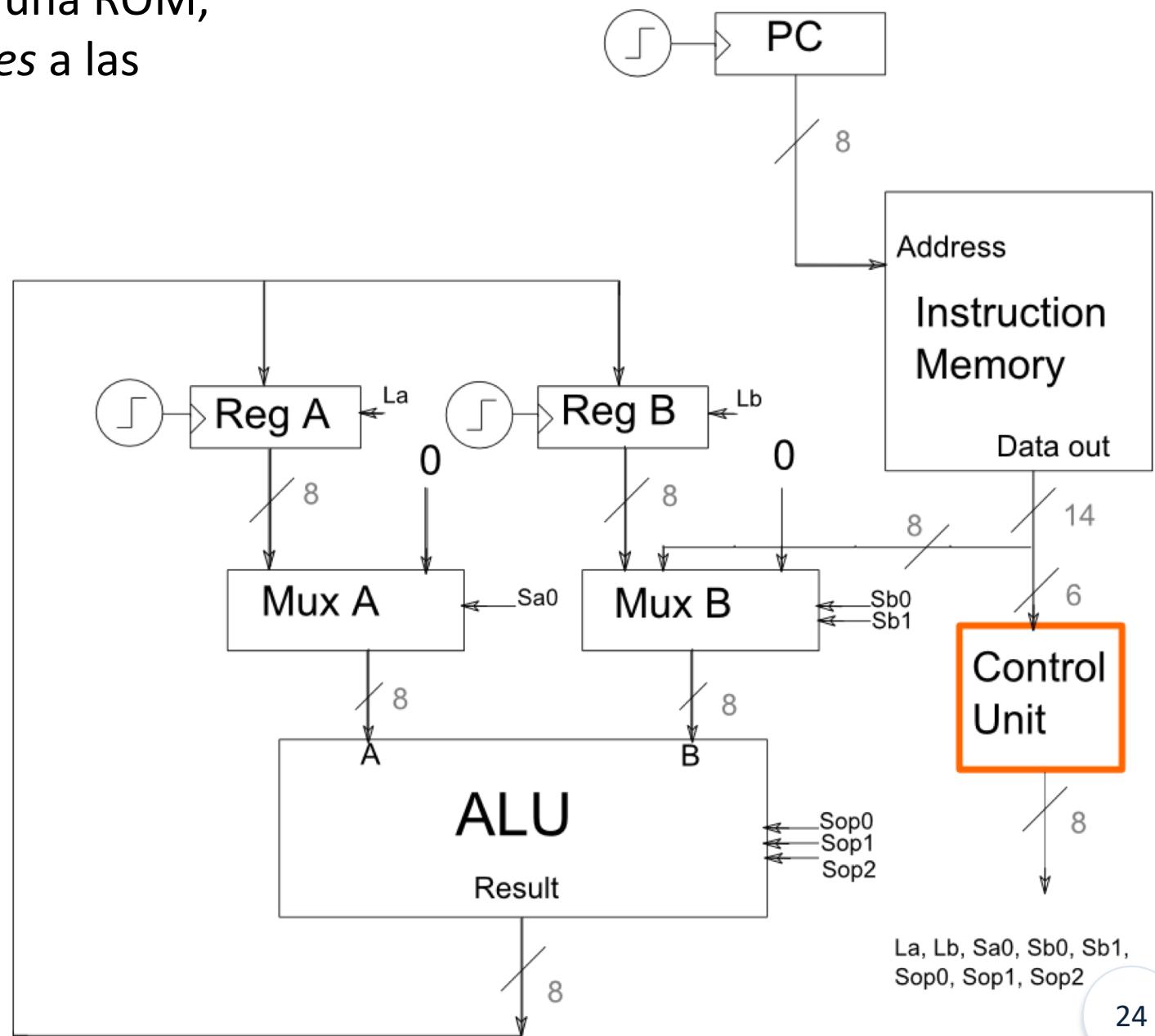
La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
1	0	1	0	0	0	0	0	A=B
0	1	0	1	1	0	0	0	B=A
1	0	0	0	1	0	0	0	A=Lit
0	1	0	0	1	0	0	0	B=Lit
1	0	0	0	0	0	0	0	A=A+B
0	1	0	0	0	0	0	0	B=A+B
1	0	0	0	1	0	0	0	A=A+Lit
1	0	0	0	0	0	0	1	A=A-B
0	1	0	0	0	0	0	1	B=A-B
1	0	0	0	1	0	0	1	A=A-Lit
1	0	0	0	0	0	1	0	A=A and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=notA
0	1	0	0	0	1	0	0	B=notA
1	0	0	0	1	1	0	0	A=notLit
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	1	1	1	0	A=shift left Lit
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A
1	0	0	0	1	1	1	1	A=shift right Lit

Usamos opcodes,
cada uno asociado a
una instrucción:

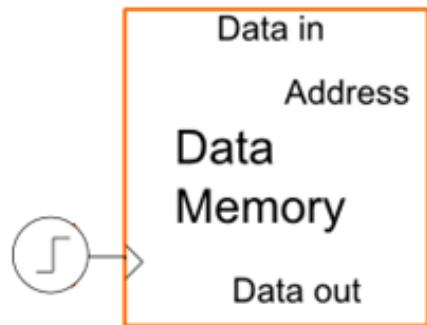
- es decir, numeramos (en binario) las instrucciones correlativamente y usamos estos números como identificadores de las instrucciones

Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
000000	1	0	1	0	0	0	0	0	A=B
000001	0	1	0	1	1	0	0	0	B=A
000010	1	0	0	0	1	0	0	0	A=Lit
000011	0	1	0	0	1	0	0	0	B=Lit
000100	1	0	0	0	0	0	0	0	A=A+B
000101	0	1	0	0	0	0	0	0	B=A+B
000110	1	0	0	0	1	0	0	0	A=A+Lit
000111	1	0	0	0	0	0	0	1	A=A-B
001000	0	1	0	0	0	0	0	1	B=A-B
001001	1	0	0	0	1	0	0	1	A=A-Lit
001010	1	0	0	0	0	0	1	0	A=A and B
001011	0	1	0	0	0	0	1	0	B=A and B
001100	1	0	0	0	1	0	1	0	A=A and Lit
001101	1	0	0	0	0	0	1	1	A=A or B
001110	0	1	0	0	0	0	1	1	B=A or B
001111	1	0	0	0	1	0	1	1	A=A or Lit
010000	1	0	0	0	0	1	0	0	A=notA
010001	0	1	0	0	0	1	0	0	B=notA
010010	1	0	0	0	1	1	0	0	A=notLit
010011	1	0	0	0	0	1	0	1	A=A xor B
010100	0	1	0	0	0	1	0	1	B=A xor B
010101	1	0	0	0	1	1	0	1	A=A xor Lit
010110	1	0	0	0	0	1	1	0	A=shift left A
010111	0	1	0	0	0	1	1	0	B=shift left A
011000	1	0	0	0	1	1	1	0	A=shift left Lit
011001	1	0	0	0	0	1	1	1	A=shift right A
011010	0	1	0	0	0	1	1	1	B=shift right A
011011	1	0	0	0	1	1	1	1	A=shift right Lit

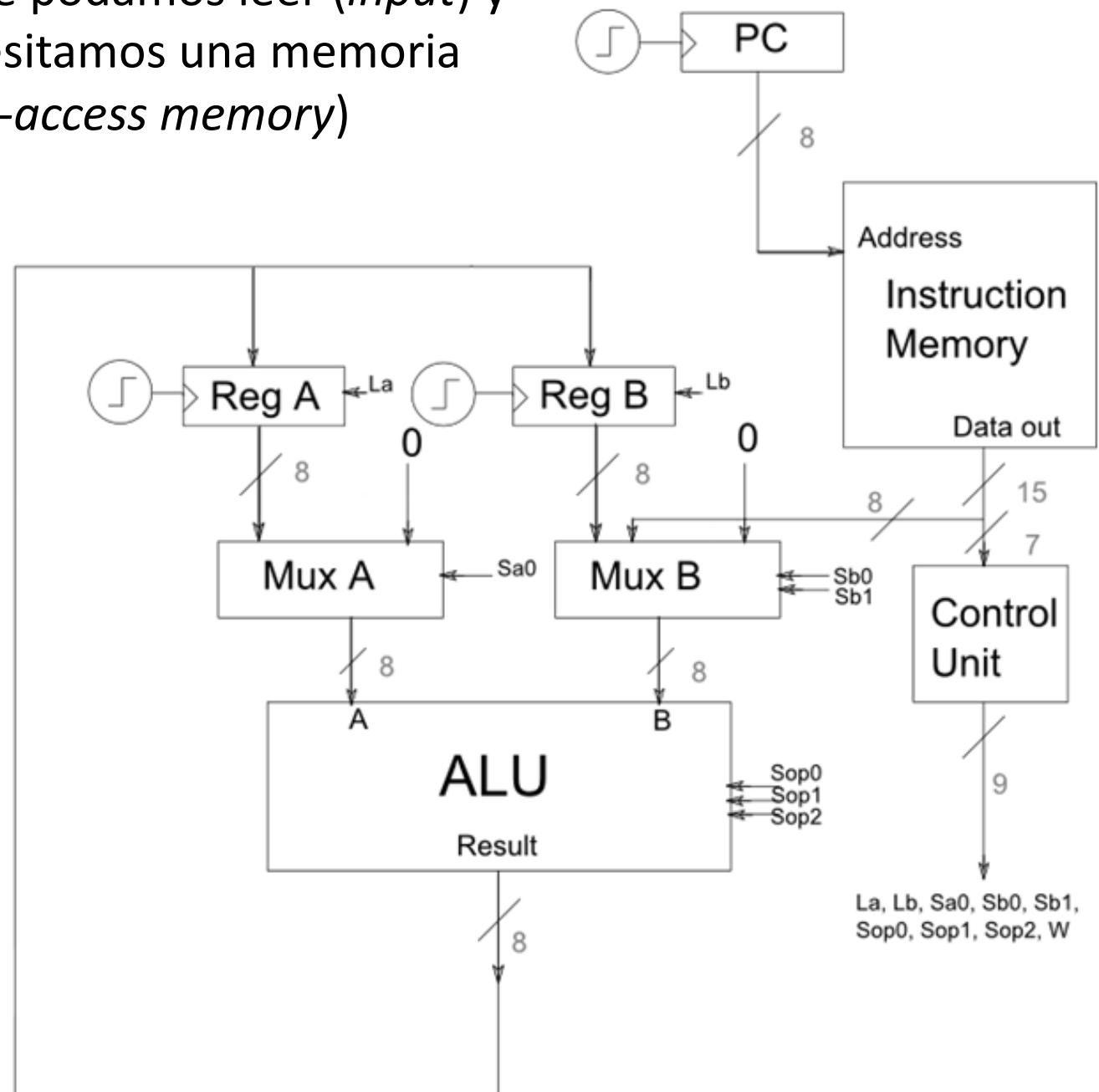
Una **unidad de control** (**CU**), implementada en una ROM, traduce los *opcodes* a las señales de control



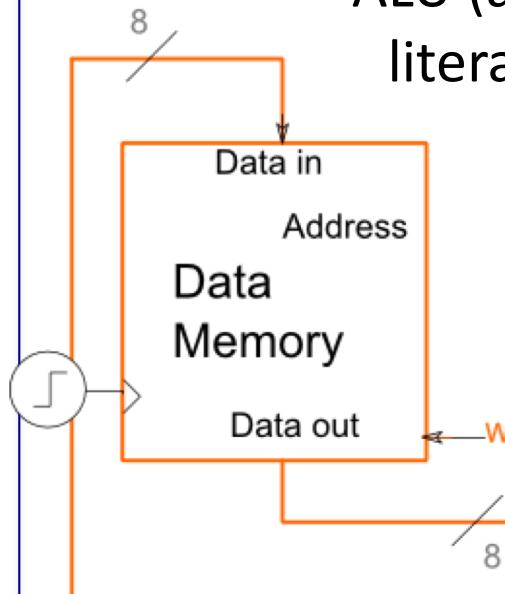
Para poder manejar una (mucho) mayor cantidad de datos, que podamos leer (*input*) y escribir (*output*), necesitamos una memoria de tipo **RAM** (*random-access memory*)



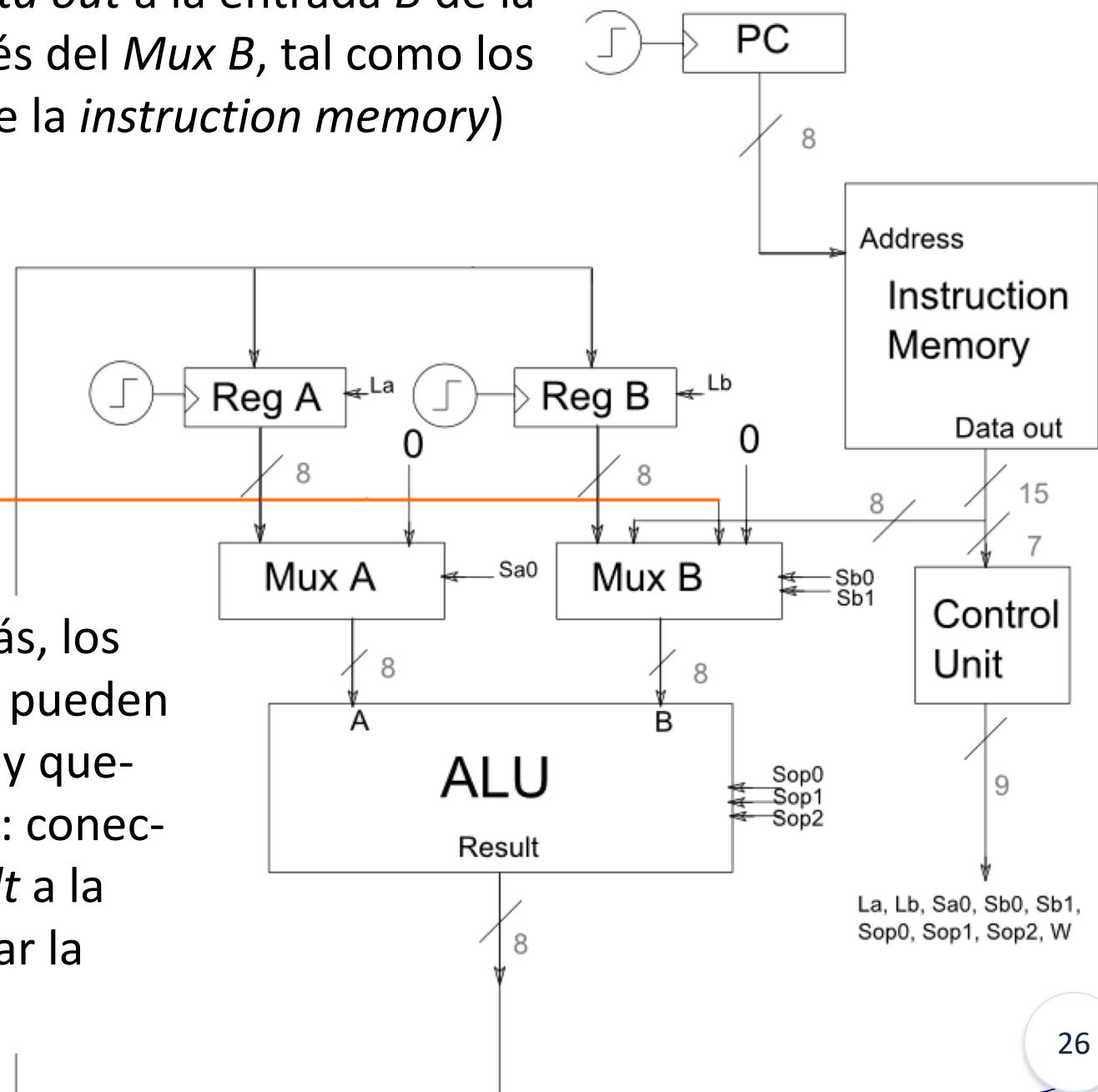
Esta *data memory* es similar a la *instruction memory*, pero además de la entrada *Address* tiene una entrada *Data in*



Los datos de la *data memory* van desde la salida *Data out* a la entrada *B* de la ALU (a través del *Mux B*, tal como los literales de la *instruction memory*)



... pero ahora, además, los resultados de la ALU pueden ir a la *data memory*, y quedar almacenados allí: conectamos la salida *Result* a la entrada *Data in*; notar la señal de control *w*



La dirección de la palabra de la *data memory* que queremos leer ($w = 0$) o escribir ($w = 1$) se puede especificar de dos maneras:

